

# NetSlices: Scalable Multi-Core Packet Processing in User-Space

Tudor Marian  
Cornell University

Ki-Suh Lee  
Cornell University

Hakim Weatherspoon  
Cornell University

## ABSTRACT

Modern commodity operating systems do not provide developers with *user-space* abstractions for building high-speed packet processing applications. The conventional raw socket is inefficient and unable to take advantage of the emerging hardware, like multi-core processors and multi-queue network adapters. In this paper we present the NetSlice operating system abstraction. Unlike the conventional raw socket, NetSlice tightly couples the hardware and software packet processing resources, and provides the application with control over these resources. To reduce shared resource contention, NetSlice performs domain specific, coarse-grained, spatial partitioning of CPU cores, memory, and NICs. Moreover, it provides a streamlined communication channel between NICs and user-space. Although backward compatible with the conventional socket API, the NetSlice API also provides batched (multi-) send / receive operations to amortize the cost of protection domain crossings. We show that complex user-space packet processors—like a protocol accelerator and an IPsec gateway—built from commodity components can scale linearly with the number of cores and operate at 10Gbps network line speeds.

## 1. INTRODUCTION

Extensible and programmable router support is becoming more important within today’s experimental networks [21, 27, 28, 40, 47]. Indeed, general purpose packet processors enable the rapid prototyping, testing, and validation of novel protocols. For example, OpenFlow [36] evolved quickly into a mature specification, and was able to do so by leveraging highly extensible NetFPGA [33] forwarding elements. Moreover, the OpenFlow specification is currently being incorporated into silicon fabric by enterprise grade router manufacturers. Such extensible router support seamlessly enables the deployment of functionality that is currently implemented by network providers through special purpose network middle-boxes, like as protocol accelerators and performance enhancement proxies [42, 48, 55].

Traditionally, the tradeoff between specialized hardware packet processors and software packet processors running on general purpose commodity hardware has been, and remains still, one of high performance versus ease of pro-

grammability. The currency for packet processors is performance. More recently, several significant efforts strived to render networking hardware more extensible [29, 33, 45]. Conversely, software routers have successfully harnessed the raw horsepower of modern hardware to achieve considerably high data rates [16, 27, 34]. However, for the sake of performance, such software routers were devised to run within the kernel, at a low level immediately on top of the hardware.

Writing a packet processor on domain specific, albeit extensible, hardware is hard since the developer needs to be aware of low level issues, intricacies, and limitations. We argue that building packet processors in the kernel, even when taking advantage of elegant frameworks such as Click [30], is equally difficult. In particular, the developer does not simply learn a new “programming paradigm.” She needs to be aware of the idiosyncrasies of the memory allocator (e.g. small virtual address spaces, the limit on physically contiguous memory chunks, the inability to swap out pages), understand various execution contexts and their preemptive precedence (e.g. interrupt context, bottom half, task / user context), understand synchronization primitives and how they are intimately intertwined with the execution contexts (e.g. when an execution context is not allowed to block), deal with the lack of standard development tools like debuggers, and handle the lack of fault isolation. A bug in a conventional monolithic kernel brings the system into an inconsistent state and is typically lethal—leading at best to a crash, or worse, may corrupt data on persistent storage or cause permanent hardware component failure.

Although user-space packet processing applications could ease the development burden and provide fault isolation, the premium on performance has rendered such an option largely invalid for all but modest data rates. Packet processors running in user-space on modern operating systems (OSes) are rarely able to saturate modern networks [14, 16, 37], given that 10 Gigabit Ethernet (GbE) Network Interface Controllers (NICs) are currently a commodity. Yet the opportunity to achieve both performance, fault isolation, and programmability rests in taking advantage of multi-core processors and multi-queue NICs. However, to scale linearly with the number of cores, contention must be kept to a minimum. Conventional wisdom, and Amdahl’s law [3, 26], states that when

adding processors to a system the benefit grows at most linearly while the costs (cache coherency, memory / bus contention, serialization) grow quadratically. Unfortunately, operating systems fail to provide general-purpose abstractions for packet processing in user-space that take advantage of modern hardware transparently. For example, packet processors built with the *raw socket*—the de-facto packet processing mechanism—are unable to sustain high rates.

In this paper we report on the design and implementation of NetSlice—a new operating system abstraction that enables linear performance scaling with the number of cores while processing packets in user-space. We achieve this through an efficient raw communication channel akin to the raw socket, that leverages modern hardware. NetSlice performs *spatial partitioning* (i.e. exclusive assignment instead of time sharing) of the CPU cores, memory, and multi-queue NIC resources at coarse granularity, to aggressively reduce overall memory and interconnect contention.

NetSlice provides high performance and multi-core scalability. It tightly couples the hardware and software resources involved in packet processing. The spatial partitioning effectively offers the illusion of a battery of independent, isolated SMP machines working in parallel with little contention. At the same time, each individual NetSlice partition is designed to provide a fast, lightweight, streamlined path for packets between the NICs and the user-space raw endpoint. Moreover, the NetSlice application programming interface (API) exposes fine-grained control over the hardware resources, and provides efficient batched send / receive operations.

NetSlice is practical; it works out-of-the-box with vanilla Linux kernels running stock NIC device drivers (simply build and load NetSlice at runtime), achieving high-performance without requiring any invasive patches (e.g. it requires no new system-calls or modified zero-copy drivers). Unlike NetTap [7] and the more recent netmap [56], NetSlice does not rely on zero-copy techniques, though it could benefit from them; consequently, NetSlice is able to trivially enforce strict address space isolation, as well as provide seamless portability and usability. NetSlice is self-contained, as portable as any device driver, and easy to deploy, requiring only a simple kernel extension that can be loaded at runtime.

We show that complex user-space packet processors built with NetSlice—a protocol accelerator and an IPsec gateway—closely match the performance of state-of-the-art in-kernel RouteBricks [16] variants. Moreover, NetSlice packet processors scale linearly with the number of cores and operate at nominal 10Gbps network line speeds, vastly exceeding alternative user-space implementations that rely on the conventional raw socket. NetSlice is fundamentally different than high-speed in-kernel variants like RouteBricks since the latter does not provide fault isolation. Further, RouteBricks does not enable high-speed packet processing in user-space any more than the conventional raw socket does.

The contributions of this paper are as follows:

- We argue that the conventional abstractions (e.g. the

raw socket) are ill-suited for packet processing applications.

- We propose NetSlice—a new operating system abstraction for developing packet processors in user-space that can leverage modern hardware.
- We show that the throughput of NetSlice applications scales linearly with the number of cores, closely following the performance of state-of-the-art, in-kernel variants. NetSlice also provides fault isolation.
- NetSlice requires only a simple kernel extension which can be loaded at runtime, providing hardware independent drop-in replacement for conventional raw sockets.

The rest of the paper is structured as follows. Section 2 expands on the motivation behind user-mode packet processors. Section 3 details the NetSlice design and implementation while Section 4 presents our evaluation. Section 5 contains the related work and Section 6 concludes.

## 2. RAW SOCKETS AND MANY CORES: WHERE HAVE ALL MY CYCLES GONE?

The new reality is that software packet processors must scale with the number of cores (e.g. routing throughput should increase as the number of cores increase). This is true even for user-space packet processors. Currently, packet processors and packet capture libraries rely on the raw socket (`PF_PACKET` and `SOCK_RAW`) and BSD Packet Filter [35] (`BPF/LSF`). Unfortunately, these abstractions were designed when the ratio between single CPU performance (expressed in cycles/MIPS/MFLOPS) and network speed remained the same over time. By shifting the focus from single CPU scaling to placing many cores on the same silicon chip, the semiconductor industry has ushered in a new world in which fast networks are driven in unison by many slow cores. For example, modern 10 Gbps commodity network adaptors are commonplace, and while the number of cores per chip has been steadily increasing, single core performance has been stagnant for the past years.

The raw socket and other sister operating systems abstractions for packet processing in user-space are overly general, and in need of an overhaul. The issues stem from the fact that the entire network stack handles the raw socket in the same fashion it handles a regular endpoint (e.g. TCP or UDP) socket—essentially taking the least common denominator between the two. However, unlike TCP or UDP sockets, a raw socket is different in that it manipulates the entire traffic seen by the host. Given today’s network capabilities and the relatively slow cores, such traffic is sufficient to overwhelm a host that uses raw sockets. We argue that applications are unable to take advantage of modern hardware since:

1. The raw socket abstraction is too general and provides the user-mode application with no control over the physical resources involved in packet reception and transmission.

2. Although simple and common to all types of sockets, the socket API is largely inefficient.
3. The conventional network stack is loosely coupled. In particular, the hardware and software resources that are involved in packet processing are loosely coupled, which results in increased contention.
4. Likewise, the conventional network stack was built for the general, most common case. As a result, the path taken by a packet between the NIC and the user-space raw endpoint is unnecessarily expensive.

Engler et al. [19] have similarly argued for an “end-to-end” approach against the high cost introduced by high-level abstractions. A fixed set of high level abstractions has been known to **i)** hurt application performance, **ii)** hide information from applications, and **iii)** limit the functionality of applications. The conventional (raw) socket is such an example: it offers a single, arguably ossified, API which abstracts away the path taken by a packet between the NIC and the application, thus providing no control over the hardware resources utilized, which is why applications fail to perform.

In what follows, we will expand on the four above claims. First, the socket API does not provide tight control over the physical resources involved in packet processing. For example, the user-mode application has no control over the path taken by a packet between some NICs queue and the raw endpoint. Second, although providing a simplified interface, the socket API is largely inefficient. For example, it requires a system call for every packet send / receive operation (the asynchronous I/O interface is currently only used for file operations, since it does not support ordering—equally important for both TCP send/receive and UDP send operations).

Third, the network stack is loosely coupled. For example, the raw socket endpoint is loosely coupled with the network stack by virtue of the user-mode task it belongs to. Since processing is performed in a separate protection domain, an additional cost is incurred due to packet copies between address spaces, cache pollution, context switches, and scheduling overheads. The cost depends on the CPU affinity of the user-mode task relative to the corresponding in-kernel network stack that processed the packets in the first place. In general, there are several choices where the user-mode task may run with respect to the in-kernel network stack:

- **Same-core:** in lockstep on the same CPU with the in-kernel network stack;
- **Hyperthread:** concurrently on a peer hyperthread of the CPU that runs the in-kernel network stack;
- **Same-chip:** concurrently on a CPU that shares the Last Level Cache (LLC), e.g. L3 for Nehalem;
- **Different-chip:** concurrently on a CPU that belongs to a different packaging socket / silicon die.

The first option, i.e. same-core, is ideal in terms of cache performance, however one has to consider the cost of frequent context switches and the impedance mismatch between the in-kernel network stack running in softirq context (a type of bottom half), at a strictly higher priority than user-mode tasks, and the user-mode task. If the user-mode task is not time-shared sufficient CPU cycles to clear the socket buffers in a timely fashion, packets will be dropped.

If hyperthreads are available, the second option may be ideal. However, hyperthreads need to be simultaneous—the CPU can fetch instructions from multiple threads in a single cycle. Hyperthreads are not ideal if they work on separate data (i.e. at different physical locations in memory), since they would split all shared cache levels into half. However, if hyperthreads work on shared data, like the packets passed between a user-mode task and the in-kernel network stack, then this scenario has the potential of also reducing cache misses beyond the LLC. Alternatively, two CPUs may only share the LLC—the third option—and still reduce the number of cache misses. The final option is sub-optimal, since every packet would induce an additional LLC cache miss.

However, the kernel scheduler defaults to dynamically selecting on which CPU to run the user-mode task, constantly re-evaluating its past decision, and potentially migrating the task onto a different CPU. Although the user-space application is able to choose a CPU affinity to request on which CPUs to run, the socket interface provides no insight into what the placement should be. The traffic may have been handled by the in-kernel network stack on any of the available CPUs. Worse, the raw socket was designed to receive traffic from all queues of every NIC, traffic that is handled by all (interrupt receiving) CPUs, thus increasing the cross-core contention overhead (e.g. cache coherency, cache pollution).

Fourth, and final, the in-kernel network stack is overly general, bulky, and unnecessarily expensive. To illustrate this, consider a user-space application processing the entire traffic by means of raw sockets. For the system depicted in Figure 1(a), in order to utilize the available CPU cores, boilerplate solutions either use several raw sockets in parallel, one per process / thread, or a single raw socket and load balance traffic to several worker threads.

If several raw sockets are used in parallel, each received packet is processed by protocol handlers as many times as there are raw sockets, and a copy of the packet is delivered to each of the raw sockets. Moreover, the original packet is also passed to the default in-kernel IP layer. To implement a packet processor in user-space, an additional firewall rule is needed that instructs the kernel to needlessly drop the packet. Berkeley Packet Filters (BPF) can be installed on each raw socket in an attempt to disjointly split the traffic, however:

- BPF filters are expensive, and they scale poorly with an increase in the number of sockets [63].
- Writing non-overlapping filters for all possible traffic patterns is hard at best, and requires a priori knowledge

of traffic characteristics, not to mention the complexity of handling traffic imbalances. Filters may be installed at runtime, by reacting to the traffic patterns, however, installing filters on the fly at rates around 10Gbps is not feasible [61].

- Without understanding the NICs opaque hash function that classifies flows to queues we are unable to predict the CPU that will be executing the kernel network stack, hence filters may exacerbate interference (e.g. cache misses). Such predictions are only possible if the interrupts from queues are issued in a deterministic fashion, and if the classification function is itself deterministic. The issue is further aggravated by using NICs from different vendors, which implement different classification functions (in our experience this is true of the Intel and Myricom 10GbE NICs).

Alternatively, a single raw socket may be used to load balance and quickly dispatch traffic to several worker threads. In this scenario, there are two potential contention spots. First, between the in-kernel network stacks running on all (interrupt receiving) CPUs and the dispatch task, and second, between the dispatch task and the worker threads (we evaluate this scenario in Section 4).

### 3. NetSlice

We argue that user-mode processes need complete control over the entire path taken by packets, all the way from the NICs to the applications and back. NetSlice relies on a four pronged approach to provide an efficient OS abstraction for packet processing in user-space. First, NetSlice spatially partitions the hardware resources at coarse granularity to reduce interference / contention. Second, the NetSlice API provides the application with fine-grained control over the hardware resources. Third, NetSlice provides a streamlined path for packets between the NICs and user-space. Fourth, NetSlice exports an efficient API.

The core of the NetSlice design consists of spatial partitioning of the hardware resources involved in packet processing. In particular, we provide an array of independent packet processing execution contexts that “slice” the network traffic to exploit parallelism and minimize contention. We call such an execution context a NetSlice. Each NetSlice tightly couples all software and hardware components like NICs and CPUs—executing the in-kernel network stack and the user-mode task tightly coupled with each-other.

As network speeds have continued to increase and vendors have switched focus from individual CPU performance scaling to increasing the number of cores per chip, a single core handling traffic at line rate from a single network interface has few, if any, cycles to spare. Modern NICs attempt to address the issue by supporting in hardware multiple transmit (tx)/receive (rx) queues that can be handled in parallel by different cores. A NetSlice packet processing execution context consists of one such tx and one rx queue

per attached NIC, and two (or more) *tandem* CPUs. Importantly, a tx/rx NIC queue belongs to a single context, hence each NetSlice context can perform any interface-to-interface forwarding independently. While the NIC queues and CPU cores are resources explicitly partitioned by NetSlice, each execution context also consists of implicit resources, like a share of the physical memory, PCIe bus bandwidth, etc. The tandem CPUs share at the very least the LLC; NetSlice defaults to using hyperthreads if available. NetSlice automatically binds the tx/rx queues of each context to issue interrupts exclusively to one of the peer CPUs in the context—we call this the *k-peer* CPU; we call the other CPU(s) the *u-peer* CPU(s). The in-kernel (NetSlice) network stack executes on the k-peer CPU, while the user-mode (NetSlice) task runs simultaneously on the u-peer CPU. A NetSlice may have more than two CPUs: several threads execute in user-mode to balance the processing load between user- and kernel-space.

There are as many NetSlices as there are CPU tandems. For our experimental setup depicted in Figure 1(a), NetSlice partitions resources as depicted in Figure 1(b). Every NIC is configured with eight tx/rx queues, associating the  $i^{th}$  tx/rx queue of every NIC (e.g. NICs 0 and 1 in Figure 1(a)) with tandem pairs consisting of CPUs  $i$  (k-peer) and  $i+8$  (u-peer). Each NIC issues interrupts signaling events pertaining to the  $i^{th}$  queue to the  $i^{th}$  CPU exclusively. Through this technique, no two k-peer CPUs will handle packets on the same NIC queue, thus eliminating the costs of contention like locking, cache coherency, and cache misses. This scheme that binds NIC queues to CPUs was previously evaluated for 1Gbps NICs [8] and is the keystone to RouteBricks’ individual forwarding element scaling. (RouteBricks relies on Click [30] which uses a polling driver—the same functionality provided by the “New API” (NAPI) [41] hybrid polling in conjunction with NIC device interrupt coalescence.)

NetSlice exposes fine-grained control over the hardware resources of the entire packet processing execution context to the user-mode application. For example, it provides control over which CPU the in-kernel (NetSlice) network stack is executing with respect to the user-mode application to take advantage of the physical cache layout. The added control is key to minimizing inter-CPU contention in general, and cache misses and cache coherency penalties in particular.

Importantly, the path a packet takes through each NetSlice execution context is streamlined, bypassing the default, bulky, in-kernel general purpose network stack. NetSlice hijacks the packets at an early stage subsequent to DMA reception and before it would have been handed off to the network stack. Next it performs minimal processing while in kernel context executing on the k-peer CPU, and then passes the packets to the user-space application to be processed in overlapped (pipelined) fashion, on the u-peer CPU. Notably, on an entire NetSlice path there is a single spinlock being used per send / receive direction. The spinlock is specialized for synchronization between the communicating execution contexts, namely between a bottom half and a task context.

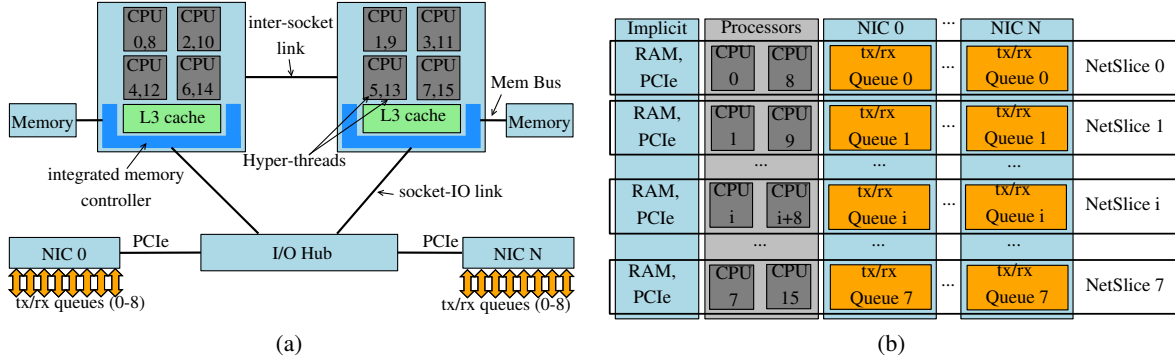


Figure 1: Nehalem cores and cache layout (a) and corresponding NetSlice spatial partitioning example (b).

While the NetSlice API provides tight control over physical resources, it also supersedes and extends the socket API while maintaining a level of backwards compatibility. Besides requiring a system call for every packet send or receive, the NetSlice API also supports batched operations to amortize the cost associated with protection domain crossings.

### 3.1 NetSlice Implementation and API

The raw NetSlice API extends the device-file interface and leverages the flexibility of the `ioctl` mechanism. User-mode libraries may use it to create a more elegant API, in the same fashion the Packet CAPture (pcap) library [51] is layered on top of the raw socket. These user-mode applications perform conventional file operations using the familiar API (`read/write/poll`) over each slice, which map to corresponding operations over the per-NetSlice data flows. For example, a conventional read operation will return the next available packet, block if no packet is available, or return `-EAGAIN` if there are no packets available and the device was opened with the `O_NONBLOCK` flag set. We implemented the NetSlice abstraction as a set of character devices with the same major number and  $N$  minor numbers—one minor number for each of the  $N$  slices. By overloading the device-file interface we gained portability since NetSlice could reside in a kernel runtime loadable module, whereas new system calls cannot.

The `ioctl` mechanism was sufficient to provide NetSlice additional control and API extensions. For example, the `NETSLICE_CPUMASK_GET` `ioctl` request returns the affinity mask of the tandem CPUs, providing the current user-mode task with fine control over the CPU it runs atop. The `NETSLICE_TX_CSUM_SET` `ioctl` request allows the user-mode application to offload TCP, IP, both or no checksum computation (alternatively, one may set a per-packet flag in the `netslice_iov`). The in-kernel NetSlice stack has the knowledge to enable hardware specific offload computation to spare CPUs from unnecessarily spending cycles.

Once the `NETSLICE_RW_MULTI_SET` `ioctl` is issued, the user-mode application may overload the `read/write` calls to send and receive an array of datagrams encoded into the parameters. Note that this is fundamentally different than

the `readv/writev` calls which can only perform scatter-gather of a *single* datagram (or packet) per call. Batched packet receive and send operations are instrumental in mitigating the overheads of issuing a system call per operation. At the same time, batching reduces per packet locking overheads, e.g. spinlock induced cycle waste and cache coherency overheads, between the user-mode task while executing system calls and the in-kernel NetSlice network stack.

Figure 2 shows an example of application code using NetSlice batched read / write for a naïve deep packet inspection tool. Commenting out lines 37 and 39, the application forwards packets behaving as a regular router. The array of buffers are passed to the read and write functions encoded in `netslice_iov` structures. The example consists of a single NetSlice (namely the first NetSlice) hence the application will only receive packets classified to be handled by the first queue of each NIC. To handle the entire traffic, the example can be easily extended to accommodate all available queues using either an equal number of threads or processes.

For outgoing packets, the routing decision is performed by default within the in-kernel NetSlice stack. However, there is an `ioctl` request (`NETSLICE_NOROUTE_SET`) that allows applications to instruct NetSlice that routing will be performed in user-space (by encoding the chosen output interface within the parameters of the write call). If the hardware decides which NIC rx queue to place the received packets onto, the software is responsible for selecting an outbound NIC queue to transmit packets on. For the conventional network stack, the core kernel or the device driver is responsible with implementing this functionality. NetSlice provides a specialized classification “virtual function” that overrides driver or kernel provided hash functions (by updating the `select_queue` function pointer of `net_device` structures). The NetSlice classification function ensures that the packets which belong to a particular NetSlice context are placed solely on the tx queues associated with the context. Unlike the classification functions provided by the device drivers (e.g. the Myricom `myri10ge` driver provides the `myri10ge_select_queue` function) or the kernel’s default `simple_tx_hash`, the NetSlice classification function is cheaper, consisting only of three load operations, one

```

1: #include "netslice.h"
2:
3: struct netslice_iov {
4:     void *iov_base;
5:     size_t iov_len; /* capacity */
6:     size_t iov_rlen; /* returned length */
7:     int flags; /* selective per-packet operations */
8: } iov[IOVS];
9:
10: struct netslice_rw_multi {
11:     int flags;
12: } rw_multi;
13:
14: struct netslice_cpu_mask {
15:     cpu_set_t k_peer, u_peer;
16: } mask;
17:
18: fd = open("/dev/netslice-1", O_RDWR);
19:
20: rw_multi.flags = MULTI_READ | MULTI_WRITE;
21: ioctl(fd, NETSLICE_RW_MULTI_SET, &rw_multi);
22: ioctl(fd, NETSLICE_CPUMASK_GET, &mask);
23: sched_setaffinity(getpid(), sizeof(cpu_set_t),
24:     &mask.u_peer);
25: for (i = 0; i < IOVS; i++) {
26:     iov.iov_base = malloc(MTU_LARGE);
27:     iov.iov_len = MTU_LARGE;
28: }
29: if (mlockall(MCL_CURRENT) < 0)
30:     EXIT_FAIL_MSG("mlockall");
31:
32: for (;;) {
33:     ssize_t cnt, wcnt = 0;
34:     if ((cnt = read(fd, iov, IOVS)) < 0)
35:         EXIT_FAIL_MSG("read");
36:
37:     for (i = 0; i < cnt; i++)
38:         /* iov_rlen marks bytes read */
39:         scan_pkg(iov[i].iov_base, iov[i].iov_rlen);
40:     do {
41:         size_t wr_iovs;
42:         /* write iov_rlen bytes */
43:         wr_iovs = write(fd, &iov[wcnt], cnt-wcnt);
44:         if (wr_iovs < 0)
45:             EXIT_FAIL_MSG("write");
46:         wcnt += wr_iovs;
47:     } while (wcnt < cnt);
48: }

```

**Figure 2: One NetSlice batched read/write example.**

arithmetic, one bitwise mask operation, and no conditional branches (by contrast, `simple_tx_hash` has three conditional statements).

Instead of a character device, we could have implemented NetSlice by extending the socket interface with a new socket type (e.g. `SOCK_RAW`). However, the current approach enabled us to seamlessly commandeer received packets immediately after reception. A new `PF_PACKET` socket does not curtail the default network stack, nor does it prevent the kernel from performing additional processing per packet (e.g. pass packets through all relevant protocol handlers).

## 3.2 Discussion

NetSlice does not rely on zero-copy techniques, unlike prior work for which zero-copy was the keystone in boosting the performance of I/O and network paths [17, 49]. Instead, NetSlice copies each packet once between user- and kernel-space, trading off CPU cycles in exchange for flexibility and portability. The reason we can afford to make this tradeoff is because modern NUMA (non-uniform memory access) architectures that replaced the Front-Side-Bus with point-to-point interconnects can be CPU bound when processing packets [16]; with the caveat that cache coherency overhead and cache misses can be kept sufficiently low [11]. As we’ll show in Section 4, NetSlice minimizes these overheads and achieves linear scalability with the number of cores, while maintaining the cost of packet copies constant per CPU. The cost is roughly a cache miss for the first load plus the time it takes to copy the remaining bytes which the hardware prefetching already brought in the LLC. This works to our advantage, since CPU cycles and even entire cores are easier to scale than interconnect and bus capacities: Moore’s Law currently results in an increase in cores, and the goal is to fully utilize each core. Further, commodity NICs are expected to support an increasing number of queues, since they are also used to support virtualization.

NetSlice leverages modern hardware to render zero-copy an orthogonal issue, less pivotal for performance, which can be supported in the future if so desired. By avoiding zero-copy, NetSlice gains added portability and usability. Currently, NetSlice comprises of a single runtime loadable module that works out-of-the-box with vanilla Linux kernels running stock NIC device drivers.

By contrast, zero-copy techniques, like memory mapping NIC DMA rings or the NIC’s entire address space into user-space [7, 56] are not only invasive, they also break isolation. New device drivers may have to be written and supported, and kernels may have to be extended or modified accordingly. For example, the kernel scheduler will have to ensure that the user-mode task controlling a NIC is scheduled sufficient CPU time and is not preempted for long continuous periods [9, 15] of time, or packets are dropped (when the DMA rings fill up). Moreover, current OSes lack support for delivering events to user-space in a timely fashion—mechanisms like `epoll` or `kqueue` / `kevent` do not currently feature interrupt delivery. Myricom’s MX-10G [39] is one such technology that provides zero-copy drivers. However, the MX-10G drivers are intended for TCP/IP and UDP/IP communication endpoints, and do not support interface-to-interface forwarding, the most basic functionality of a router or general purpose packet processor. In fact, the MX-10G drivers do not provide support out-of-the-box for two different NICs at the same time on the same machine.

Likewise, NetSlice is also orthogonal to the large body of past work that relocated the networking stack into user-space [7, 20, 23, 50, 59]. For example, user-space networking may very well be built on top of NetSlice, although we did not yet implement the network stack encapsulation for replacing endpoint sockets. In our experience, conventional TCP and UDP sockets still perform sufficiently well, to date. Moreover, given that a typical host may have an arbitrarily large number of concurrent TCP and UDP connections, it

is not clear that user-space networking, even built over NetSlice, would perform better than the current network stack since it would require efficient inter-process-communication for de-multiplexing and multiplexing traffic between user-mode applications. Nevertheless, we plan to investigate NetSlice support for endpoint sockets in the future.

## 4. EVALUATION

We evaluated software packet processors running NetSlice and compared them against the state-of-the-art user-space and in-kernel equivalent implementations. We have ported packet processors to run over RouteBricks [16] forwarding elements, as well as to run in user-space using the pcap library [51]. Pcap is implemented on top of the conventional raw (PF\_PACKET) sockets. We also linked the pcap applications with Phil Wood’s libpcap-mmap library [60], which uses the memory mapping functionality of PF\_PACKET sockets, known as PACKET\_MMAP. (PF\_RING [52] sockets are roughly an alternative implementation of the PACKET\_MMAP approach. Further, PF\_RING sockets require an invasive patch that alters the core-kernel codebase, unlike the readily available PACKET\_MMAP sockets or NetSlice.) A kernel that is built with the PACKET\_MMAP support copies each packet onto a circular buffer mapped into user-space before optionally adding it to the socket’s queue. The user-space application can poll the arrival of new packets and receive them without the cost of issuing additional system calls. (The PACKET\_MMAP support does not implement a zero-copy receive technique, a packet is copied the same number of times as with a traditional socket.) The NetSlice batched receive operation achieves the same net effect, however unlike NetSlice batched transmit, PACKET\_MMAP sockets do not offer equivalent support for outbound packets. During our experiments, we set the circular buffer size to the value that yielded the best performance, incidentally it was the maximum value. (For our machines, the PCAP\_MEMORY=max request yielded a 1.93GB circular buffer.) We did not compare against the libipq / libnetfilter\_queue [43] packet redirection mechanism since it consistently crashed at high data rates (the netlink sockets it relies on proved to be inadequate for high data rates).

NetSlice consists of 1814 lines of kernel module code and 2981 lines of user-space applications—a router, an IPsec gateway (839 lines of AES ports), and an implementation of the Maelstrom [5] protocol accelerator.

Our evaluation answers the following questions:

- What is the performance of NetSlice with respect to the state-of-the-art, for both routing and IPsec?
- What is the performance breakdown of the NetSlice techniques? To quantify this scenario, we funnel all traffic to be handled by a single NIC queue: there is no interference from extraneous CPUs and NIC resources, and we are able to quantify, in isolation:
  - The benefit of streamlining packet paths;

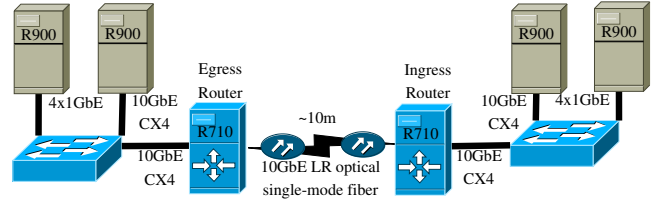


Figure 3: Experimental evaluation physical topology.

- The NetSlice performance with respect to possible peer CPUs placement;
- The benefit of NetSlice batched operations;
- NetSlice added latency and CPU usage.
- How does NetSlice scale with the number of cores?
- Can complex packet processors built with NetSlice deliver the advertised performance increase?

### 4.1 Experimental Setup

We deployed a testbed topology as depicted in Figure 3, with four Dell PowerEdge R900 machines serving as end-hosts that generate and receive traffic. The traffic is aggregated by two Cisco Catalyst 4948 series switches before being routed through a pair of identical Dell PowerEdge R710 machines, which we refer to as the egress and ingress routers. The egress and the ingress routers run various packet processor variants, like NetSlice, RouteBricks, or pcap.

Each R900 machine is a four socket 2.40GHz quad core Xeon E7330 (Penryn) with 6MB of L2 cache and 32GB of RAM—the E7330 is effectively a pair of two dual core CPUs packaged on the same chip, each with 3MB of L2 cache. By contrast, the R710 machines are dual socket 2.93GHz Xeon X5570 (Nehalem) with 8MB of shared L3 cache and 12GB of RAM, 6GB connected to each of the two CPU sockets. The Nehalem CPUs support hardware threads, or hyperthreads, hence the operating system manages a total of 16 processors. Each R710 machine is equipped with two Myri-10G NICs, one CX4 10G-PCIE-8B-C+E NIC and one 10G-PCIE-8B-S+E NIC with a 10G-SFP-LR transceiver. Figure 1(a) depicts the R710 internal structure with two NICs.

The egress router is connected to the ingress router via a 10 meter single-mode fiber optic patch cable, and each router is connected to the corresponding switch through a 6 meter CX4 cable. Two of the R900 machines are each equipped with an Intel 82598EB 10-Gigabit CX4 NIC, while the other two R900 machines are connected to the switches through all of their four Broadcom NetXtreme II BCM5708 Gigabit Ethernet NICs. We use the additional R900 machines, although the egress and ingress routers only have one 10GbE connection on each side, since a single R900 machine with a 10GbE interface is unable to receive (in the best configuration) more than roughly 5Gbps worth of MTU size (1500 byte packets) traffic. The packet rate (pps) for the R710 router with the Myricom 10GbE NIC is roughly the same for small (64 byte) and MTU size packets. The same observation applies for the R900 client with the Intel 10GbE NIC.



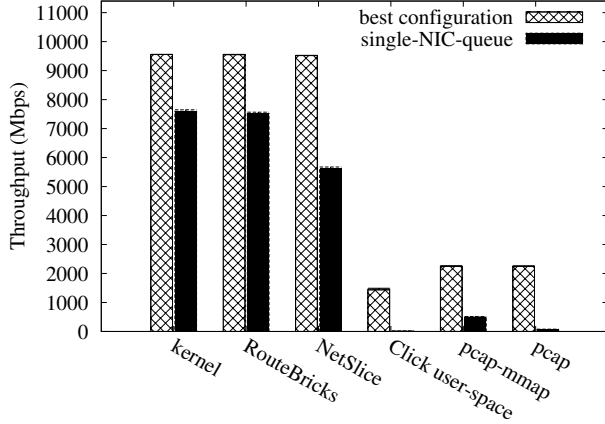


Figure 4: Packet routing throughput.

RouteBricks altered the NIC driver to increase the packet rate by performing DMA transfers of small packets in a single transaction on the PCIe bus. We have not implemented this feature yet—it is not clear such batched DMA transfers is possible on our Myricom NICs.

Unless specified otherwise, we generate traffic between the R900 machines with Netperf [44] that consists of MTU size UDP packets at line rate (10Gbps). The machines run the Linux kernel version 2.6.28-17; we use the myri10ge version 1.5.1 driver for the Myri-10G NICs and the ixgbe version 2.0.44.13 driver for the Intel NICs. Both drivers support NAPI and are configured with factory default interrupt coalescence parameters. To enable RouteBricks, we modified the myri10ge driver to work in polling mode with Click (we used Linux kernel version 2.6.24.7 with Click, the most recent version supported).

All values presented are averaged over multiple independent runs, between as low as eight and as high as 32 runs; the error bars denote standard error of the mean and are always present, although most of the time they are sufficiently small to be virtually invisible.

## 4.2 Forwarding / Routing

Figure 4 shows the UDP payload throughput for the most basic functionality—packet routing with MTU size packets. We compare the NetSlice implementation with the default in-kernel routing, a RouteBricks implementation, and with the best configurations of pcap user-space solutions. Utilizing all NIC queues and all CPUs, NetSlice forwards packets at nominal line rate (roughly 9.7Gbps for MTU packet size and MAC layer overhead), as do the kernel and RouteBricks routing. However, the best pcap variants top off at about 2.25Gbps. There is no difference between pcap and pcap-mmap, while Click user-space does in fact perform worse.

For each case, the Figure shows the additional scenario in which all traffic is handled by a single NIC queue (per available NIC). In this case, the kernel achieves 7.59Gbps, while NetSlice achieves 74% of the kernel throughput, while pcap-mmap achieves  $\frac{1}{11}$  of the throughput achieved by NetSlice, and about 7.6 times better than regular pcap. As expected,

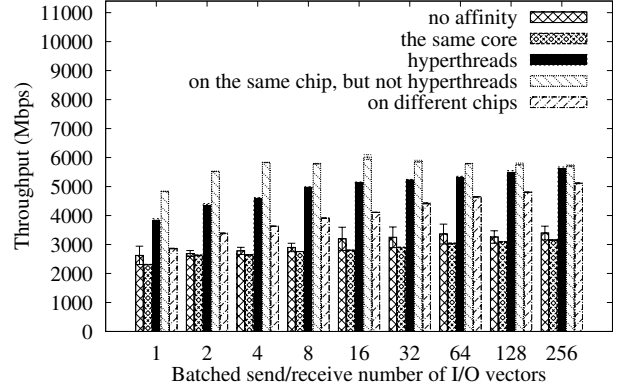


Figure 5: Routing throughput for single a NIC queue, a single NetSlice, and various u-peer CPU placements.

Configuration	Packets / $\mu s$	RTT ( $\mu s$ )
Linux kernel	1/100	242.24 $\pm$ 42.14
Linux kernel	10/100	279.48 $\pm$ 42.74
NetSlice (no batching)	1/100	255.38 $\pm$ 39.98
NetSlice (no batching)	10/100	308.10 $\pm$ 44.51
NetSlice (128-batch)	1/100	255.67 $\pm$ 40.18
NetSlice (128-batch)	10/100	301.33 $\pm$ 42.16

Table 1: Round-trip-time (RTT) between the end-hosts.

in-kernel variants perform better since routing is performed at an early stage, and less CPU work (zero-copy forwarding) is expended per dropped packet [38].

The take-away is that the NetSlice kernel to user-space communication channel is highly efficient, even when a single channel is used (here two CPUs and one NIC queue per NIC). Moreover, using more than a single NetSlice easily sustains line rate—currently, our clients are not able to generate more than 10Gbps worth of MTU-size packet traffic.

Next, we evaluate the importance of the u-peer CPU placement. User-space processing takes place on the u-peer CPU as part of the spatial partitioning that isolates individual NetSlices. Here we used a single NetSlice to stress one communication channel that handles all traffic in isolation. Since only two tandem CPU cores and one NIC queue per NIC are utilized, the experiment only accounts for direct interference (like cache coherency, cache misses due to pollution) within a single NetSlice. Additional indirect interference is expected in the general case, however, the NetSlice spatial partitioning was designed precisely to keep such interference to a minimum. Figure 5 shows the throughput given various core placement choices and the number of I/O vectors used for batched operations. There are several key observations. First, if the user-mode task does not use the CPU affinity as instructed by NetSlice, the default choice made by the OS scheduler is suboptimal. Moreover, the high error bars imply that the kernel does not attempt to perform smart task placement. The Linux scheduler is primitive in that it typically moves a task on the runqueue of a different CPU only if the current CPU is deemed congested.

The second observation is that using the same CPU for



Datarate (Mbps)	CPU usage (% of single CPU)		
	Total	k-peer	u-peer
1000.9	31.34 ± 1.00	1.90 ± 0.30	29.44 ± 0.73
2001.8	63.49 ± 1.39	12.41 ± 0.72	51.09 ± 0.68
3002.7	100.38 ± 0.88	24.56 ± 1.61	75.82 ± 0.92
4003.2	102.28 ± 2.69	14.29 ± 3.41	87.99 ± 0.72
5003.1	103.40 ± 2.78	17.94 ± 2.90	85.46 ± 1.36

**Table 2: CPU usage: One NetSlice (2 CPUs) forwarding.**

both in-kernel and user-space processing performs the worst—there are simply not enough cycles to counter the excessive overheads introduced by the context switches. Additionally, there is an impedance mismatch between the task context and the in-kernel processing that is performed in a softirq context and is of strictly higher priority than the task, i.e. the task is not scheduled enough cycles. This setting is complicated further by the kernel’s per-CPU `ksoftirqd` threads that are spawned to act as rate-limiters during receive-live-lock scenarios [38].

The third observation is that same-chip and hyperthread placement outperform the scenario in which the user-space processing happens on a different chip. This is consistent with the memory hierarchy—i.e. accessing the shared L3 cache is faster than accessing data over the QuickPath inter-socket link. However, note that the gap between same-chip and different-chip data access decreases considerably with the increase in the number of I/O vectors. This is likely because batching increases code and data locality, and hardware optimizations like pre-fetching and pipelined processing are in effect. Batched processing also improves the performance of user-space processing on the hyperthread, however to a lesser extent than same-chip placement, presumably because the hardware threads still contend for functional units (e.g. ALUs) within the (shared) physical core.

The best case is when the peer CPUs are on the same chip yet are not hyperthreads. However, the Figure shows the scenario in which a single NetSlice is used, hence only the peer CPUs are utilized, all the remaining cores are idle. In the general case, such a placement choice is only viable when there is a lower number of NetSlices than there are available CPUs. By default, NetSlice performs user-space processing on the sibling hyperthread, if one is available—having two sibling hyperthreads work on different NetSlices would split the cache levels (higher than the LLC) into half.

Figure 5 also shows the performance increase due to NetSlice batching. For the default peer CPU placement (i.e. sibling hyperthreads) we observe a 46.2% increase in aggregate throughput from singleton send / receive to 256 batched I/O vectors shuttled between user-space and the kernel in a single operation, even though the kernel uses the fast system call processor instructions (e.g. `SYSENTER`).

In summary, the forwarding throughput of a single NetSlice is eleven times larger than the best pcap. Of that, batching provides a 46.2% throughput increase, peer CPU placement provides a 78.3% throughput increase when batching is used and 66.6% increase with no batching, while the stream-

lined path of packets (with no batching or peer CPU placement) provides a 4.5 times throughput increase over pcap-mmap. This coarse break-down does not reveal the subtle interaction between NIC queues nor the cross core, cross PCIe bus, and cross QuickPath interconnect interference.

Table 1 shows the additional latency introduced by a single NetSlice forwarding element. The experiment shows the round-trip-time (RTT) between an R900 end-host and the Ingress Router while traffic flows through the Egress Router (Figure 3). The Table depicts the Egress Router performing standard in-kernel forwarding, and forwarding through NetSlice with batching both disabled and enabled (128 I/O vectors). The table shows two scenarios, one in which the sender issues packets at a steady rate of one every 100 $\mu$ s and a second in which the sender issues 10 packets in rapid succession every 100 $\mu$ s. The two-way latency introduced by NetSlice is 19 $\mu$ s on average (at most 28 $\mu$ s), which is half the standard deviation of the reported RTTs. The latency introduced by NetSlice is in fact smaller than the effects of NIC interrupt coalescence (IC) and NAPI, two ubiquitous techniques that have been universally adopted to the detriment of latency (e.g. the myri10ge driver defaults the `rx-usecs` IC parameter to 75 $\mu$ s and does not compile without NAPI).

Table 2 shows the NetSlice CPU utilization while forwarding traffic through a single NIC queue, for increasing input data rates. As expected, the CPU utilization increases with the data rate, however, the increase is less sharp for rates greater than 3Gbps. This is due to batching which, while seldom used for rates less than 3Gbps, is required to forward packets at increasingly higher data rates, as we previously showed in Figure 5.

### 4.3 IPsec

Next we evaluate a CPU intensive packet processing task, namely IPsec encryption with 128 bit key (typically used by VPNs). We implemented AES encryption in Cipher-block Chaining (CBC) [54] mode of operation. Our experiments focused on steady-state performance, hence the key establishment protocol is not evaluated. We use IPsec to evaluate how NetSlice scales with the number of cores. IPsec accelerators typically need all the CPU cycles they can spare and two NetSlices proved sufficient to forward all the 10Gbps MTU-size traffic that our testbed was able to generate.

Ideally, NetSlice should only trail RouteBricks by a constant factor (per CPU) due to the cycles spent performing an additional copy per packet and the overhead of protection domain crossings (system calls). Figure 6 shows that NetSlice does scale linearly with the number of CPUs, closely following RouteBricks. The NetSlice throughput is roughly 8% less than that of RouteBricks. RouteBricks tops off at 9157Mbps, about 600Mbps shy of nominal line rate. NetSlice tops off at about 8513Mbps. We expect both NetSlice and RouteBricks to continue to scale linearly given more cores. By contrast, the user-space variants using pcap scale poorly and are unable to take advantage of the current tech-

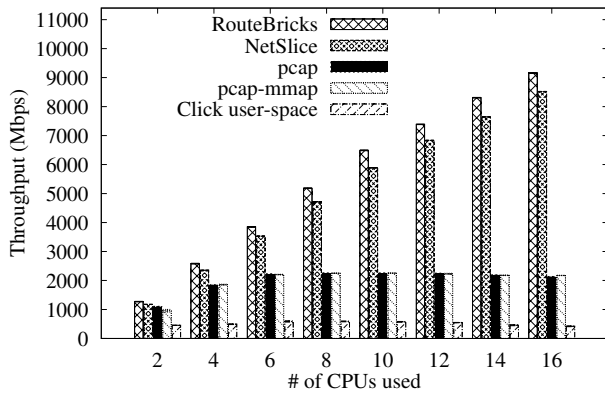


Figure 6: IPsec throughput scaling with cores.

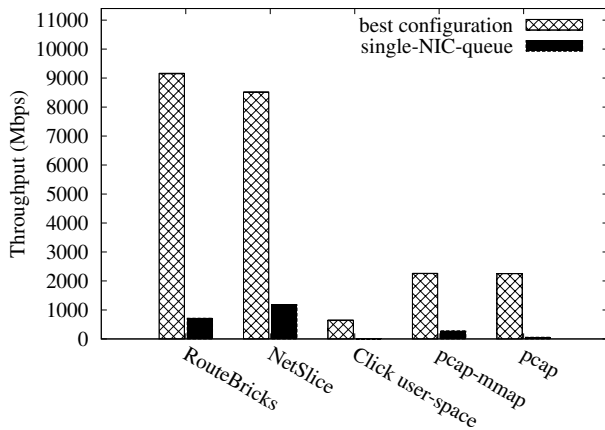


Figure 7: IPsec throughput, all vs. single NIC queue.

nology trend towards placing many independent cores on the same silicon die. (The Figure reports on the best user-space pcap variants with a dispatch thread load balancing packets to threads bound to CPUs exclusively.)

Figure 7 shows IPsec throughput results for the best configurations of NetSlice, RouteBricks, and pcap user-space solutions along with the additional scenario in which all traffic is handled by a single NIC queue. First, notice that the pcap variants top off at about 2258Mbps in the common case and perform poorly when traffic is handled by a single NIC queue. As with routing, the pcap-mmap outperforms conventional pcap in the latter scenario and NetSlice, like RouteBricks, vastly outperforms the user-space variants. NetSlice also achieves better throughput than RouteBricks since all traffic that is routed to a single NIC queue is handled by RouteBricks with a single CPU in kernel mode, whereas NetSlice handles it with a pair of CPUs, one running in kernel-mode and one running the user-mode task.

The take-away is that NetSlice scales with the number of available cores as good as the in-kernel RouteBricks implementation does. By contrast, user-space variants that use conventional raw sockets scale poorly and are comprehensively outperformed by NetSlice and RouteBricks during a CPU intensive task like IPsec.

## 5. RELATED WORK

It has been well known that large scale cache coherent, and potentially NUMA, multiprocessors require careful operating system design, or else bottlenecks prevent the systems from reaching their performance potential. Indeed, operating systems like Tornado/K42 [31, 58] have been carefully designed to minimize contention by clustering and replicating key kernel data structures, and by employing intricate scheduling algorithms that, for example, take NUMA locality into account.

More recently, there have been several research efforts that aimed at redesigning the OS from the ground up in order to effectively exploit the emerging and now ubiquitous multi-core architectures. Corey [10] is an ExoKernel-like OS within which shared kernel data structures and kernel intervention are kept to a minimum, while applications are given explicit control over the sharing of resources. This allows the Corey kernel to perform finer grained locking of highly accessed data structures, like process memory regions. The Barrelfish research operating system [6] explores how to structure the OS as a distributed system in order to best utilize future multi- and many-core, potentially heterogeneous systems. Similarly, the Helios [46] operating system tackles building and tuning applications for heterogeneous systems through satellite kernels. Satellite kernels export a uniform set of OS abstractions across all CPUs and communicate one with another by means of explicit message passing instead of relying on a cache coherent memory system. The Tessellation OS [32] introduces a “nano-visor” to enforce strict spatial and temporal resource multiplexing between library OSes. To ensure resource isolation, the Tessellation OS envisions hardware support for resources that have been traditionally hard to share, like caches and memory bandwidth.

Like the Tessellation OS, NetSlice performs spatial partitioning of the CPU, memory, and multi-queue NIC resources at coarse granularity. However, the NetSlice partitioning is domain specific, and the performance isolation need not be strongly enforced, instead it is implicit by the design of the NetSlice abstraction itself.

Historically, there have been a large number of zero-copy user-space network stacks proposed [7, 17, 23, 49, 50, 52, 56, 59]. Their general approach was to eliminate the OS involvement on the communication path, and virtualize the NIC while providing direct, low-level access to the network. Some of these approaches relied on hardware support, for example, U-Net [59] and its commercial successor VIA [4, 18] required a communications co-processor capable of demultiplexing packets into user-space buffers, and an on-board MMU (Memory Management Unit) to perform RDMA (Remote DMA) [2]. The former is not available on typical Ethernet NICs, moreover, IOMMUs are currently unable to handle page faults [22]. Other systems [12, 13, 17] rely on virtual memory and page protection techniques, however, on demand memory mapping of shared buffers is tricky and can

be unnecessarily expensive. As a result, such techniques are yet to be adopted by mainstream kernels.

NetSlice is orthogonal to prior work that placed the network stack in user-space. Further, it does not use zero-copy techniques since they were not necessary (see Section 3.2) and they would have prevented portability.

Software routers Achilles' heel has been, and continues to be, the low performance with respect to their hardware counterparts. Nevertheless, recent efforts, like RouteBricks [16], have shown that modern multi-core architectures and multi-queue NICs are well suited for building low-range software routers, albeit in kernel-space. RouteBricks relies on a cluster of PCs fitted with Nehalem multi-core CPUs and multi-queue NICs, connected through a k-degree butterfly interconnect. Packets are forwarded / routed at aggregate rates of 24.6Gbps per PC, however, the interconnect routing algorithm introduces packet re-ordering. The PacketShader [25] software router takes RouteBricks further by providing an entire framework for general-purpose packet processing that utilizes the Graphics Processing Unit (GPU).

Internally, RouteBricks uses the Click [30] modular router, an elegant framework for building functionality from smaller building blocks arranged in a flow graph. However, Click is aimed at building routers and does not easily express general packet processing; e.g. it cannot support global state that extends across building blocks.

In general, software routers are implemented within the kernel [24, 57], early in the network stack and below the (raw) socket interface. Full blown software routers like RouteBricks [16] may require distributed coordination algorithms to decide interconnect forwarding paths [1]. By contrast, NetSlice provides support for user-space implementation of individual packet processing units, independent of interconnects. Complex packet processing logic, like rule-based forwarding [53], or the distributed coordination in RouteBricks may be seamlessly built using NetSlice (NetSlice does not re-order packets).

NetSlice can be used to implement the XORP [62] open source routing platform, or to provide rapid prototyping of OpenFlow [36] forwarding elements. For example, the current NetFPGA [33] reference implementation is limited to four 1GbE interfaces (the recently launched NetFPGA-10G supports four 10GbE interfaces), whereas NetSlice is only limited by the number of CPUs and PCIe connections a commodity server can support. Moreover, developers need not have intimate Verilog knowledge, or worry about details such as gateway real-estate.

The new Threaded NAPI (TNAPI) `PF_RING` [52] support improves on the raw socket by creating one virtual NIC per receive queue and capturing traffic from each virtual NIC with a different user-space thread (somewhat similar to NetSlice's spatial partitioning). In general, such packet capture techniques are only optimized for the receive path, ignoring the transmission, which means that they provide no support for efficient interface-to-interface forwarding, which is the

most basic software router / packet processor functionality.

## 6. CONCLUSION

The end of CPU frequency scaling is ushering in a world of slow cores and fast networks. This paper introduced the NetSlice operating system abstraction that enables building scalable packet processors in user-space. NetSlice tightly couples the hardware and software packet processing resources by performing domain specific, coarse-grained, spatial partitioning of CPU cores, memory, and NIC resources. NetSlice also provides the application with control over these resources. On top of each resource partition, NetSlice superimposes independent streamlined communication channels to shuttle packets between NICs and user-space and bypass the default network stack. While it is backward compatible with the conventional socket API, the NetSlice API also provides batched send / receive operations to amortize the cost of protection domain crossings. Further, NetSlice is portable, working with existing device drivers. We demonstrate NetSlice by showing that complex user-space packet processors can scale linearly with the number of cores and operate at nominal 10Gbps line speeds.

## 7. REFERENCES

- [1] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. *SIGCOMM Comp. Comm. Rev.* 38, 4 (2008), 63–74.
- [2] AMD. I/O Virtualization Specification, 2007.
- [3] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring)* (1967), pp. 483–485.
- [4] BALAJI, P., SHIVAM, P., AND WYCKOFF, P. High performance user level sockets over gigabit ethernet. In *Cluster Comp.* (2002).
- [5] BALAKRISHNAN, M., MARIAN, T., BIRMAN, K., WEATHERSPOON, H., AND VOLLSET, E. Maelstrom: Transparent error correction for lambda networks. In *NSDI* (2008).
- [6] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09*.
- [7] BLOTT, S., BRUSTOLONI, J., AND MARTIN, C. NetTap: An Efficient and Reliable PC-Based Platform for Network Programming. In *Proceedings of OPENARCH* (2000).
- [8] BOLLA, R., AND BRUSCHI, R. Pc-based software routers: high performance and application service support. In *PRESTO* (2008).
- [9] BOS, H., DE BRUIJN, W., CRISTEA, M., NGUYEN, T., AND PORTOKALIDIS, G. FFPF: Fairly fast packet filters. *OSDI '04*.
- [10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: an operating system for many cores. In *OSDI '08*.
- [11] BROSE, E. ZeroCopy: Techniques, Benefits and Pitfalls.
- [12] BRUSTOLONI, J. C., AND STEENKISTE, P. Effects of buffering semantics on I/O performance. *OSDI* (1996).
- [13] BRUSTOLONI, J. C., STEENKISTE, P., AND BRUSTOLONI, C. User-Level Protocol Servers with Kernel-Level Performance. In *Proc. of IEEE Infocom Conference* (1998).

- [14] CROWLEY, P., FLUCZYNSKI, M. E., BAER, J.-L., AND BERSHAD, B. N. Characterizing processor architectures for programmable network interfaces. In *Supercomputing* (2000).
- [15] DE BRUIJN, W., AND BOS, H. Beltway buffers: Avoiding the os traffic jam. In *INFOCOM* (2008).
- [16] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP* (2009).
- [17] DRUSCHEL, P., AND PETERSON, L. Fbufs: a high-bandwidth cross-domain transfer facility. *SIGOPS OS Rev.* (1993).
- [18] DUNNING, D., REGNIER, G., MCALPINE, G., CAMERON, D., SHUBERT, B., BERRY, F., MERRITT, A. M., GRONKE, E., AND DODD, C. The Virtual Interface Architecture. *Micro* (1998).
- [19] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR, J. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95*.
- [20] GANGER, G. R., ENGLER, D. R., KAASHOEK, M. F., BRICEÑO, H. M., HUNT, R., AND PINCKNEY, T. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.* 20, 1 (2002), 49–83.
- [21] GENI. Global Environment for Network Innovations.
- [22] GEOFFRAY, P. A Critique of RDMA. *High-Perf. Comp.*, 2006.
- [23] GEOFFRAY, P., PRYLLI, L., AND TOURANCHEAU, B. BIP-SMP: high performance message passing over a cluster of commodity SMPs. In *Supercomputing* (1999).
- [24] GUO, D., LIAO, G., BHUYAN, L. N., LIU, B., AND DING, J. J. A Scalable Multithreaded L7-filter Design for Multi-Core Servers. In *Proceedings of ANCS* (2008).
- [25] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: A GPU-Accelerated Software Router. In *SIGCOMM* (2010).
- [26] HILL, M. D., AND MARTY, M. R. Amdahl's law in the multicore era. *IEEE COMPUTER* (2008).
- [27] HUANG, J.-C., MONCHIERO, M., AND TURNER, Y. Ally: OS-Transparent Packet Inspection Using Sequestered Cores. In *Proceedings of ANCS* (2011).
- [28] INTERNET2. <http://www.internet2.edu/>.
- [29] JUNIPER. Open IP Service Creation Program (OSCP).
- [30] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *TOCS* (2000).
- [31] KRIEGER, O., AUSLANDER, M., ROSENBERG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A., AND UHLIG, V. K42: building a complete operating system. In *EuroSys* (2006).
- [32] LIU, R., KLUES, K., BIRD, S., HOFMEYER, S., ASANOVIĆ, K., AND KUBIATOWICZ, J. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar* (2009).
- [33] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA—An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of MSE* (2007).
- [34] MA, Y., BANERJEE, S., LU, S., AND ESTAN, C. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *SIGMETRICS '10*.
- [35] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX* (1993).
- [36] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (2008), 69–74.
- [37] MIKLAS, A. G., SAROIU, S., WOLMAN, A., AND BROWN, A. D. Bunker: A privacy-oriented platform for network tracing. In *Proceedings of NSDI* (2009).
- [38] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.* 15, 3 (1997).
- [39] MYRICOM. <http://www.myri.com/scs/download-mx10g.html>.
- [40] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM* (2009).
- [41] NAPI. <http://linuxfoundation.org/en/Net:NAPI>.
- [42] NETEQUALIZER. <http://netequalizer.com/>.
- [43] NETFILTER. [libipq/libnetfilter.queue](http://www.netfilter.org/projects/libnetfilter.queue/).
- [44] NETPERF. <http://netperf.org/>.
- [45] NETWORKWORLD. Cisco opening up IOS. <http://www.networkworld.com/news/2007/121207-cisco-ios.html>, 2007.
- [46] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., AND HUNT, G. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP* (2009).
- [47] NLR. National LambdaRail. <http://www.nlr.net/>.
- [48] PACKETLOGIC. <http://proceranetworks.com/>.
- [49] PAI, V., DRUSCHEL, P., AND ZWAENPOEL, W. IO-Lite: a unified I/O buffering and caching system. *ACM TOCS '00*.
- [50] PAKIN, S., LAURIA, M., AND CHIEN, A. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Supercomputing* (1995).
- [51] PCAP. [tcpdump / libpcap](http://www.tcpdump.org/).
- [52] PF.RING. <http://www.ntop.org/PF-RING.html>.
- [53] POPA, L., EGI, N., RATNASAMY, S., AND STOICA, I. Building extensible networks with rule-based forwarding. In *OSDI* (2010).
- [54] RFC 3602. The AES-CBC Cipher Algorithm and Its Use with IPsec, 2003.
- [55] RIVERBED. <http://www.riverbed.com/solutions/optimize/>.
- [56] RIZZO, L. netmap: a novel framework for fast packet i/o. In *USENIX ATC '12*.
- [57] SCHULTZ, M. J., WUN, B., AND CROWLEY, P. A Passive Network Appliance for Real-Time Network Monitoring. In *Proceedings of ANCS* (2011).
- [58] UNRAU, R. C., KRIEGER, O., GAMSÄ, B., AND STUMM, M. Experiences with locking in a NUMA multiprocessor operating system kernel. In *OSDI* (1994).
- [59] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: a user-level network interface for parallel and distributed computing. *Proceedings SOSP* (1995).
- [60] WOOD, P. [libpcap-mmap](http://public.lanl.gov/cpw/).
- [61] WU, Z., XIE, M., AND WANG, H. Swift: A fast dynamic packet filter. In *Proceedings of NSDI* (2008).
- [62] XORP. <http://xorp.org>.
- [63] YUHARA, M., BERSHAD, B., MAEDA, C., ELIOT, J., AND MOSS, B. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX ATC* (1994).